Semantica di Copia e Spostamento

Programmazione Avanzata e Parallela
2022/2023

Alberto Casagrande

Copiare gli Oggetti

Copiare un'oggetto della classe `Umano` avrà significati diversi nel caso in cui `Umano` rappresenti:

- un disegno di un essere umano
- un insieme di dati
- un essere umano completo

Copiare gli Oggetti

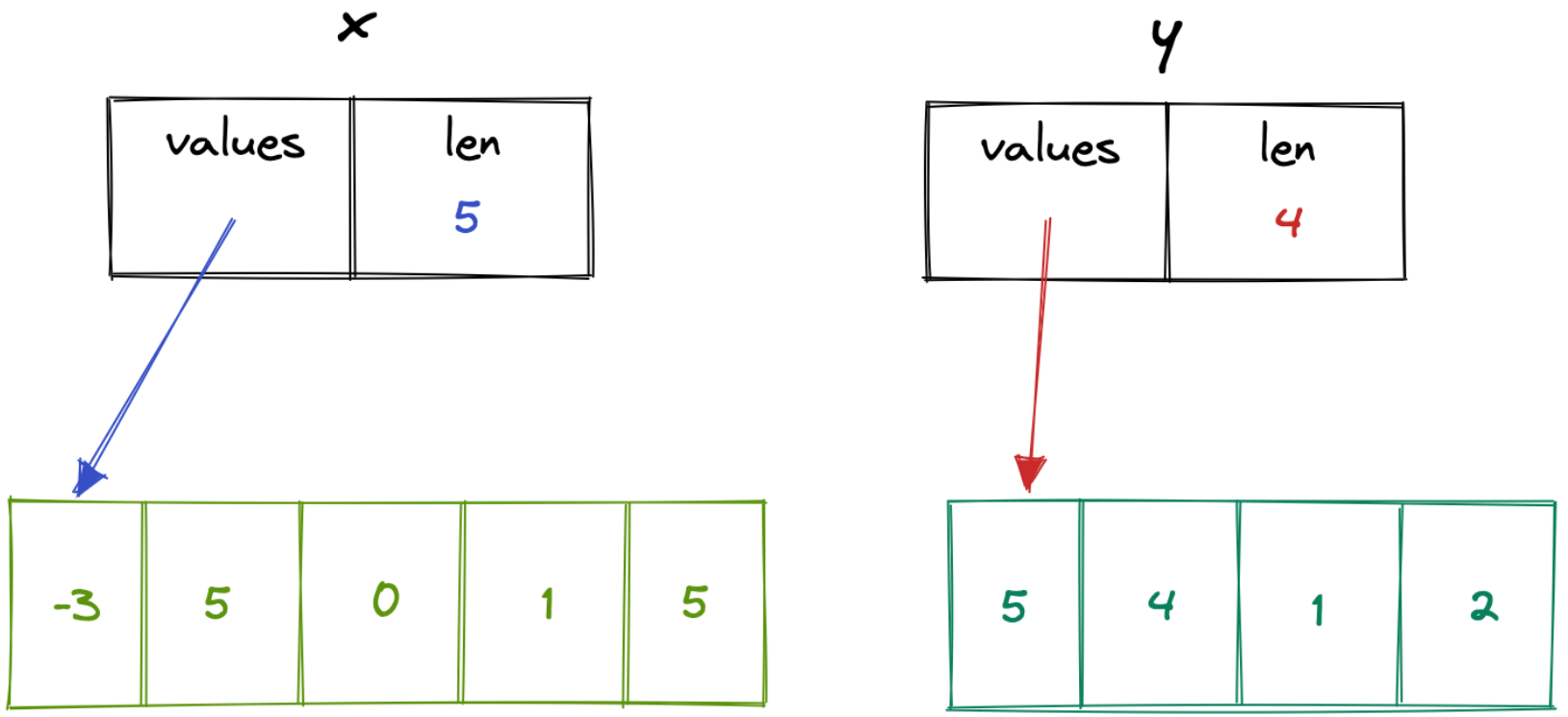
Copiare un'oggetto della classe `Umano` avrà significati diversi nel caso in cui `Umano` rappresenti:

- un disegno di un essere umano
- un insieme di dati
- un essere umano completo

Il significato di copiare un oggetto e, per esensione, il metodo con cui si effettua la copia sono chiamati **semantica della copia**

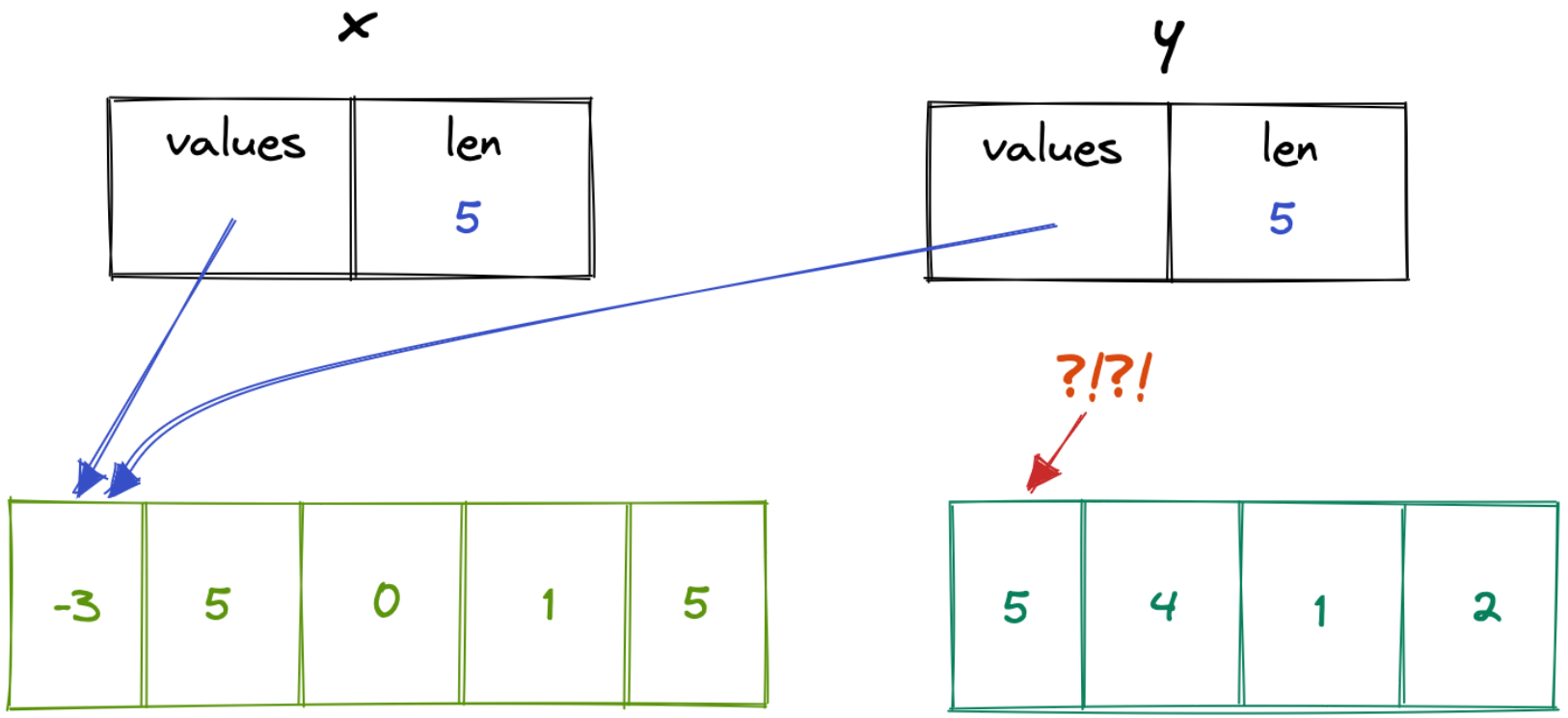
La Semantica Implicita della Copia

Quando copiamo `x` (`Array<int>`) in `y`...

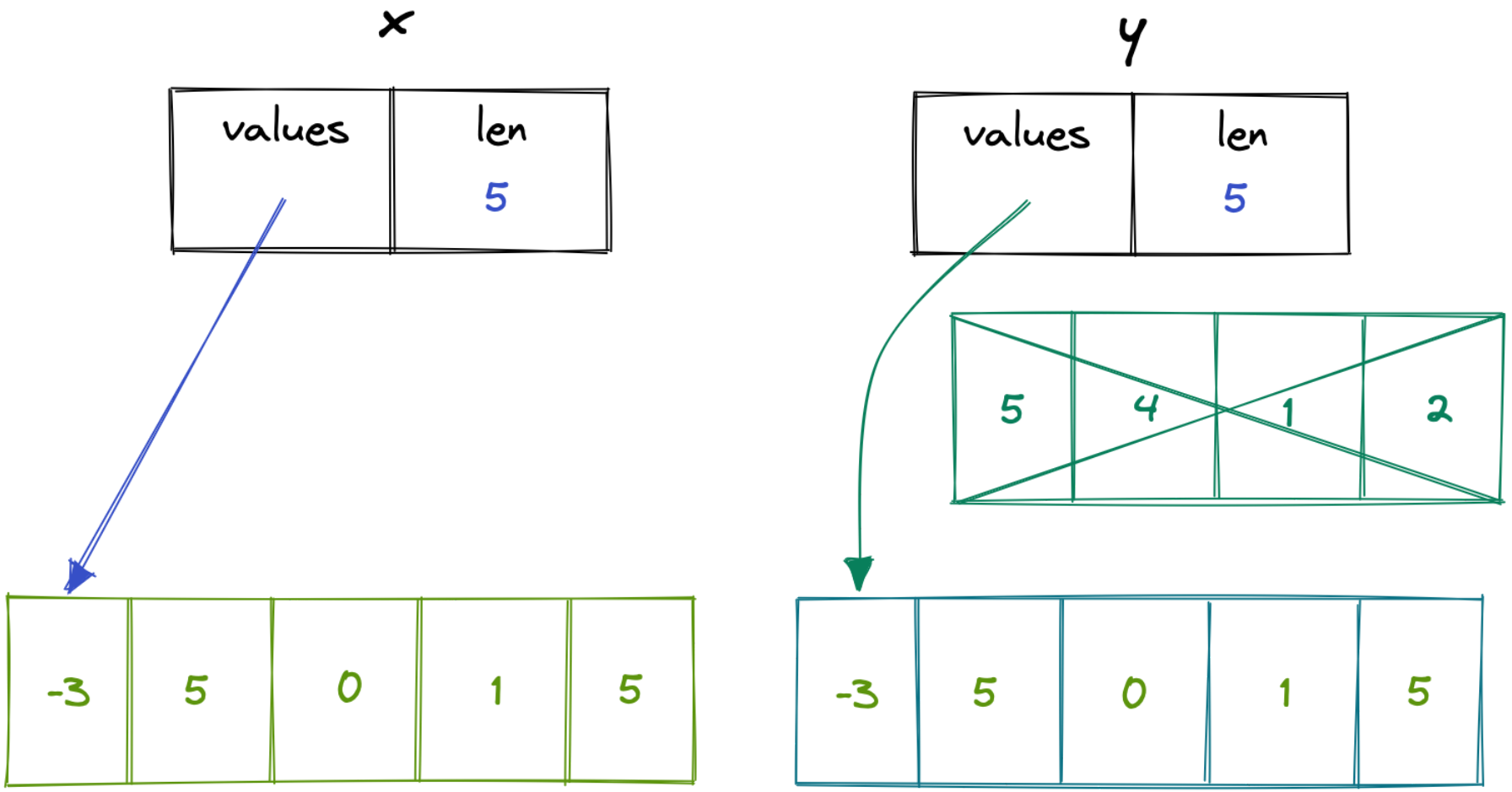


La Semantica Implicita della Copia

Quando copiamo `x` (`Array<int>`) in `y`, copiamo i membri



La Semantica delle Copia che Vorremmo



Dobbiamo Sovraccaricare...

1. il costruttore di copia, i.e., `Array(const Array<T>&)`

```
Array(const Array<T>& orig)
 : Array<T>(orig.size())  // creo un array della dimensione opportuna
{
  // copio i valori da orig.values a this->values
  std::copy(orig.values, orig.values+size(), values);
}
```

Dobbiamo Sovraccaricare...

1. il costruttore di copia, i.e., `Array(const Array<T>&)`
2. l'operatore di copia, i.e., `operator=(const Array<T>&)`

```
Array<T>& operator=(const Array<T>& orig) {
 if (this != &orig)  {  // se non sono lo stesso oggetto
    delete[] values;   // de-alloco l'array dei valori dell'oggetto *this

    len = orig.size();          // aggiorno la lunghezza dell'Array
    values = new T[size()];     // alloco un nuovo array per i valori
    std::copy(orig.values, orig.values+size(), values); // copio i valori
 }

 return *this;
}
```

Semantica della Copia e Complessità

Se stiamo copiando un contenitore di dati, e.g., un `Array` o un albero:

- la semantica *implicita* di copia prende tempo $O(1)$
- la semantica di copia che copia tutti i dati (*deep copy*) costa $O(n)$

La copia profonda dovrebbe essere utilizzata ogni volta che una funzione restituisce un oggetto...

```
Array<T> a(3,4), b(3,-3);
auto c = a+b;          // a+b restituisce un Array che deve essere copiato
                       // interamente
```

Tuttavia...

... non sembra essere necessaria...

```
Array<T> a(3,4), b(3,-3);
auto c = a+b;          // il risultato di a+b è copiato in c e poi distrutto

auto d{a+b};           // il risultato di a+b è copiato in d e poi distrutto
```

Tuttavia...

... non sempre sembra essere necessaria...

```
Array<T> a(3,4), b(3,-3);
auto c = a+b;          // il risultato di a+b è copiato in c e poi distrutto

auto d{a+b};           // il risultato di a+b è copiato in d e poi distrutto
```

Se copiamo il risultato di una valutazione (*rvalue*), e.g., `a+b`, dopo la copia il risultato non è più necessario

Perché fare una copia profonda?

Semantica dello Spostamento

Distinguiamo la copia di un oggetto riferito da un *lvalue* (`<type>&`) da quella di un oggetto riferito da un *rvalue* (`<type>&&`)

Sovraccarichiamo

1. il costruttore di spostamento, e.g., `Array(Array<T>&&)`
2. l'operatore di spostamento, e.g., `operator=(Array<T>&&)`

Il Costruttore di Spostamento per `Array<T>`

Per i soli *rvalue*, copiamo i membri e "resettiamo" `orig`

```
Array(Array<T>&& orig)
: len{orig.len}, values{orig.values}  // copio i membri di orig in *this
{
    orig.len = 0;                     // la lunghezza di orig è impostata a 0
    orig.values = nullptr;            // orig.values non punta più alla memoria
}                                     // allocata dinamicamente
```

L'Operatore di Spostamento di `Array<T>`

Per i soli *rvalue*, de-allochiamo l'array dei valori e copiamo i membri

```
Array<T>& operator=(Array<T>&& orig) {  // solo per rvalue
    delete[] values;                    // de-allochiamo lo spazio per i valori di *this
    values = orig.values;               // copiamo orig.values in values
    orig.values = nullptr;              // rimuoviamo il puntatore da orig.values

    len = orig.len;                     // aggiornamo la lunghezza di *this

    return *this;
}
```

L'Operatore di Spostamento di `Array<T>`

Per i soli *rvalue*, de-allochiamo l'array dei valori e copiamo i membri

```
Array<T>& operator=(Array<T>&& orig) {  // solo per rvalue
    delete[] values;                    // de-allochiamo lo spazio per i valori di *this
    values = orig.values;               // copiamo orig.values in values
    orig.values = nullptr;              // rimuoviamo il puntatore da orig.values

    len = orig.len;                     // aggiornamo la lunghezza di *this

    return *this;
}
```

Sovraccaricando `std::swap` possiamo semplificare il codice

Semplificare il Codice

Ci fa comodo sovraccaricare anche `std::swap`

```
template<typename T>
void std::swap(Array<T>& A, Array<T>& B) {
    std::swap(A.len, B.len);
    std::swap(A.values, B.values);
}

Array<T>(Array<T>&& orig): Array(0) {  // costruisco un array di lunghezza 0
    std::swap(*this, orig);           // scambio l'array corrente con orig
}                                     // orig è ora l'array vuoto

Array<T>& operator=(Array<T>&& orig) {
    std::swap(*this, orig);       // this->values è assegnato a orig.values
}                                 // e verrà de-allocato dal distruttore
```

Altri Vantaggi dello Spostamento

Supponiamo che `operator+` implementi la somma tra vettori:

- quanti oggetti vengono creati/copiati dall'istruzione `a+b+c+d`?
- quanti oggetti sono veramente necessari per valutare `a+b+c+d`?

Sovrascriviamo `operator+` per evitare che venga costruito un nuovo oggetto a ogni somma

Da *lvalue* a *rvalue*

Possiamo ottenere un *rvalue* da un *lvalue* usando `std::move`

```
Vector a, b, c, d;

...

a = std::move(b);       // l'oggetto riferito da b è ora riferito da a
                        // da ora in poi b non mai usato: la semantica
                        // dello spostamento poter

...
```